Valid Data Groups

Adam Lyon

22 July 2004

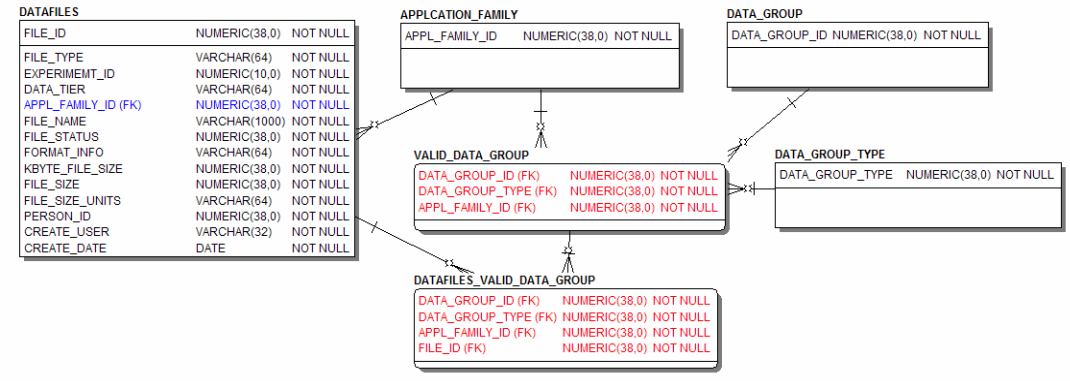
ABSTRACT: This document describes the utility of Valid Data Groups (VDGs), a new addition to the SAM Schema currently in production. The VDG schema is explained and then several use cases are given. VDGs offer a general solution to the "processing problem" that is being currently discussed. This solution for the specific use case of merging is given and then generalized to other processing activities.

1 Introduction

Valid Data Groups are a new concept to the SAM meta-data catalog that allow one to group files together by assigning them to one or more lists. Each list is a valid data group (hereafter abbreviated VDG). A VDG is identified by a name, a type, and an application family. VDGs have applications for a wide variety of use cases and may be important for some problems facing SAM today.

2 Valid Data Group Schema

An almost complete picture of the VDG schema is shown below. This schema is already in the production Oracle Database.



The import table is *DATAFILES_VALID_DATA_GROUP*. This table specifies what VDG a file belongs to (and a file can be in more than one row and so belong to more than one group). The other fields specify the VDG by connections to the VDG name, the VDG type and the application family. The *VALID_DATA_GROUP* table merely enforces a list of known VDGs to avoid

mistakes. In the DATA_GROUP and DATA_GROUP_TYPE tables, descriptions (not shown) are paired with their IDs.

3 Initial Motivation

VDGs were introduced by Wyatt after an ugly reco reprocessing effort at DØ. Some files that had been processed by reco were known to be bad and needed to be reprocessed. The problem was that the version of the reco application for this reprocessing did not change. Therefore, it was difficult to distinguish the old bad files from the new good files. Furthermore, the old bad files were only bad in one small aspect and were still useable by many groups. And in fact many people wanted to be able to process these files during the reprocessing – that is analyze the new good files that had come out of the reprocessing along with the old files that had not yet been reprocessed.

A mechanism was needed to separate the old files from the new files so that datasets could be created on each. What was decided was to change the conventions of the reco application version. The old files had reco application version of "p13.01" [not sure if the numerals are right, but that's irrelevant] while the new files had reco application version of "r13.01".

This mechanism of introducing a new fake version number was extremely unsatisfying as it broke the reco version convention. Users would have to explicitly remember this "r13.01" version and use it in their dataset definitions. If a user forgot this version, they would miss analyzing a significant amount of the data (though now all of this data has been reprocessed again and the special version identifier is no longer needed).

The fact that an easy and elegant solution to this problem could not be found revealed a deficiency in the SAM meta-data DB schema and lead to the invention of VDGs.

3.1 How Valid Data Groups would have solved the problem

The problem here is that there were files that were different, but shared the same standard meta-data. The introduction of the fake application version solved this problem by forcing the meta-data to be different, but in a very unnatural way. VDGs aim to solve the problem by assigning files to lists. One could imagine a list, or VDG, of the old, not yet processed files and a list of the new reprocessed files. When an old file is reprocessed and the new file is stored in SAM, its entry in the "old" list is removed, and an entry for the new file is added to the "new" list.

Users wanting to analyze the union of old, not yet processed files and the new reprocessed files would create a data set that simply combined the two VDGs. If the VDGs were maintained automatically, then the user would automatically get the right files. All the user would have to remember is the names of those VDGs, instead of obscure version numbers.

Once the reprocessing was complete, the old files could be marked with bad content status. The new fixed files would be available using the standard application versions. The VDGs would no longer be needed and could be removed from the database.

4 An alternative to VDGs – file tags

An alternative to VDGs is to tag the files themselves with new meta-data. That is create a new file attribute that contains one or more tags (or bit flags – the mechanism is immaterial). To solve the problem above, could imagine this attribute would be a flag that was "processed" or "unprocessed". All of the old files would be marked as unprocessed at the start of the reprocessing. As a file was reprocessed and the new file put in SAM, the tag on the old file would be changed from "unprocessed" to "processed". Then some new tag would have to be invented for the new file, like "reprocessed".

Users would then select the unprocessed tag (along with data-tier, application family, etc.) to get the not-yet-processed old files and then the "reprocessed" tag to get the new files.

4.1 Comparison to Valid Data Groups

There are some clear disadvantages to the file tags idea:

- Scalability: File tags do not scale. If multiple operations need to be kept track up (e.g. merging, skimming for group A, skimming for group B), then each would need one or more tags. Once could leave room in the schema for a sensible number of tags, but that seems artificial and no doubt we will exceed it one day. VDGs, on the other hand, scale easily. One just creates the new set of lists for each purpose. The number of lists is only limited by how much stuff the DB can hold.
- Connection to Application version and Types: The file tags would presumably be just a name or a status word attached to the file. VDGs, on the other hand, have a richer identification mechanism. One could set up VDGs with the same name but associated with different types and different application families. This could allow things like testing an operation using a "test" VDG type. Or skimming with a different application. The VDG names could hold to some standard (like "WZ

skimming") while the purpose of the VDGs could be further refined by the type and application family connection. File tags would have to be renamed for each of these purposes, leading to long and ugly names.

- Ease of use: A query to get files of a certain tag would also involve the other meta-data to identify the file (data-tier, application family, perhaps file name? etc.). For VDGs, one would only need to specify the VDG (presumably by name and if necessary, application family and type). The VDG gloms the meta-data into a list that can be obtained easily.

5 Some use cases

Below are some use cases for Valid Data Groups

5.1 Gabriele's "Bookkeeping of files for merging" use cases

These use cases come from Gabriele's mail at
http://listserv.fnal.gov/scripts/wa.exe?A2=ind0407&L=sam-design&F=&S=&P=3904

5.1.1 Monte Carlo production on Sam Grid

Monte Carlo Event production yields a set of files that are temporally stored (the "unmerged set"). These files are then merged into a smaller set of larger sized files. Problems:

- The merged files and the unmerged files contain the same data, and in fact the same events. Therefore their meta-data are nearly identical. Using data-tier to separate the files is an overloading of the data-tier principle and is undesirable. The sets of files only differ in how events are distributed within them – the events themselves are the same.
- One needs to keep track of what unmerged files have already been processed. The current proposal is to check if the unmerged files have children, and if so then assume they have been processed. This works if one can guarantee that no one will run over the unmerged files before the merge is made. One could argue that the unmerged files are private and no one except for the merger would have access to them, but this restriction may not be acceptable. One could imagine instances where groups would want to immediately skim large general MC samples before or while the merging was taking place. In that case, relying on the existence of children will not work because the unmerged files may have children due to the other processes.
- One also needs to know that once a file has been merged, it can be marked for disposal.

- If something bad happens to the merging process, it should be restarted easily. Therefore one has to know what files have been already merged.

Valid data groups provide an elegant and scalable method to handle this use case. For maximum flexibility, one would need three VDGs for the operation (one could get away with as little as one VDG).

- VDG A contains the list of unmerged files yet to be processed
- VDG B contains the list of unmerged files that have been processed
- VDG C contains the list of merged files produced

(Note that I have purposefully not indicated how these VDGs should be named, see section 6).

When a merged file is stored into SAM, the following would happen:

- The merged file is added into VDG C
- The unmerged files used to produce the merged files are removed from VDG A and added to VDG B

Note that one does this above operation only when a merged file is created. One could imagine moving each unmerged file from VDG A to VDG B as it is processed. But if the merger crashes the incomplete merged file is in an unknown state and must be thrown away. Following the operation above makes reproducing that merged file and continuing the merging process very easy.

This mechanism of using VDGs solves all of the problems.

- The merged and unmerged files are easily separated by querying the different VDGs.
- The files that need to be merged are always in VDG A. If the merger is stopped and needs to be restarted, one still simply uses VDG A.
- Users that want to analyze the union of merged files and remaining unmerged files would create a dataset with the union of VDG A and VDG C. The resulting list of files would always be consistent.
- Once the merge is complete, VDG B would be used to easily determine what files could be disposed. The merged files in VDG C already have the right standard meta-data and would take their rightful place among the experiment's files without any further operation. To save space, the VDGs themselves could then be discarded.

5.1.2 CDF wants to analyze files while they are being merged

This is covered in section 5.1.1.

5.1.3 Marking files that participated to a process as "processed"

This is covered in section 5.1.1. As mentioned there, producing a rescue dataset is simple.

5.1.4 Consistency

To quote Gabriele, "We would like to find a mechanism to bookkeep the status of these files similar to what sam does already e.g. for skimmed files." Not sure what is meant here. How does SAM do bookkeeping for skimmed files? Aside from the "consumed" status, it doesn't do anything special.

5.1.5 Users don't need retraining

This is a tricky issue and is one where VDGs do not help. In order to access things correctly during a merging process, the users will need to know the right VDG(s). After the merging is done, then one assumes the users would be able to go back to the standard meta-data as the unmerged files are discarded.

The proposal for handling this use case with file tags involves putting new defaults into the dataset definition maker to make the queries somewhat transparent. But what must be realized is that this task of analyzing files while their creation process is ongoing is a **special** task. It is not unreasonable to expect that some special effort would be required by those too impatient to wait for the creation process to complete. Furthermore, querying on VDGs is easy – all that is needed are the identifiers (name, perhaps type, and perhaps application family).

5.2 Extensions to Gabriele's use cases

5.2.1 Merging and skimming MC files simultaneously

Say a large, fairly general MC sample is being produced (say a generic BB-bar sample). The MC sample is so large that it needs to be skimmed to make further analysis easy (skimming means writing out events that pass certain criteria). Perhaps several skims will be made, each with different criteria. The original MC sample could be thrown away, but it is decided that since producing it was very painful, its files should be merged and stored in case additional skimming needs to be done. The requirement is that this merging be performed simultaneously with the skimming operations in order to save time.

- How does one keep track of what has been skimmed (for each skim)?
- How does one keep track of what has been merged?

Again, the easy solution is to create three VDGs for each operation. As discussed in section 5.1.1, this setup allows one to easily track what is being produced,

what has been processed and what is remaining to be processed. See section 6 for more information.

6 A general solution to the processing problem with Valid Data Groups

The problems discussed in section 5 are all similar:

- There is a group of files that need to be processed.
- One wants to keep track of while files from that group have been processed and what files from that group have not been processed.
- One wants to keep track of the new files that are produced.
- The standard meta-data are not enough to easily distinguish the new files from the old files.

The general solution is to create three VDGs. The VDGs would all have the same name (perhaps the request ID or the name of the skimming process) and application family. The VDGs would differ by their type.

- VDG of type "to-be-processed" holds the list of unprocessed files
- VDG of type "processed" holds the list of unprocessed files that are now processed
- VDG of type "final" holds the list of new files that were produced

With these VDGs, one can easily:

- Determine what has and has not been processed (VDG of type "to-be-processed")
- Create datasets combining the unprocessed and the final files (the "to-be-processed" VDG + the "final" VDG)
- Create rescue datasets (VDG of type "to-be-processed")
- Determine what files are disposable (VDG of type "processed")

If multiple operations are occurring for a set of files (skimming and merging) then each operation would have its set of three VDGs. They would differ by the VDG name (which would identify the process [e.g. "merge request 11223", "skim A request 11223", "skim B request 11223"]). Then all processes could be tracked independently and easily.

7 Other Use Cases

I can think of more use cases than I have time to write! More to come…

8 Required tools

The schema for VDGs is already in production. What is required and crucial for the success of VDGs are tools that make managing VDGs as simple,

straightforward and automated as possible. The dataset definition creator will also need updating to allow obtaining files in particular VDGs.

These tools will be needed:

- Creating a new VDG
- Adding a file or a set of files to a VDG (initiated by a sam store and/or by hand)
- Removing a file or set of files from a VDG (initiated by a sam store and/or by hand [e.g. remove parents])
- Create datasets giving a VDG identifier.
- More…?